Program Assignments, Intro. to Compiler Construction:

I. Use JavaCC or SableCC to implement a parser for the MiniJava language in appended part I. **Due: Nov. 22th**

II. Design a set of visitors which type-checks a MiniJava program and produces any appropriate error messages about mismatching types or undeclared identifiers. **Due: Dec. 13th**

III. Implement a simpler Translator with a set of visitors, to translate a MiniJava program into intermediate representation trees.

To simplify the implementation of the translator, you may do without the Ex, Nx, Cx constructors. The entire translation can be done with ordinary value expressions. This means that there is only one Ex class (without subclasses); this class contains one field of type Tree.Exp and only an unEX() method. Instead of Nx(s), use Ex(ESEQ(s, CONST(o)). For conditionals, instead of a Cx, use an expression that just evaluates to 1 or 0.

Due: Jan. 5th

Part I Appended: MiniJava Language

MiniJava is a subset of Java. The meaning of a MiniJava program is given by its meaning as a Java program. Overloading is not allowed in MiniJava. The MiniJava statement System.out.println(…); can only print integers. The MiniJava expression e.length only applies to expressions of type int[].

A1. Lexical Issues:

Identifiers: An identifier is a sequence of letters, digits, and underscores, starting with a letter. Uppercase letters are distinguished from lowercase. The SYMBOL_ID stands for an identifier.

Integer literals: A sequence of decimal digits is an integer constant that denotes the corresponding integer value. The symbols INTEGER_LITERAL stands for an integer constant.

Binary operators: A *binary operator* is one of

&& < + - *

In this appendix the symbol op stands for a binary operator.

Comments: A comment may appear between any two tokens. There are two forms of comments: One starts with /*, ends with */, and may be nested; another begins with // and goes to the end of the line.

A2. Grammar

In the MiniJava grammar, we use the notation N*, where is a nonterminal, to mean 0, 1, or more repetitions of N.

Program → MainClass ClassDecl*

MainClass → Class id { public static void main (String [] id) { Statement } }

ClassDecl → Class id { VarDecl* MethodDecl*}

→ Class id extends id { VarDecl* MethodDecl* }

VarDecl → Type id;

MethodDecl → public Type id (FormalList)

→ { VarDecl* Statement* return Exp ; }

FormalList → Type id FormalRest*

→

FormalRest → , Type id

Type → int []

→ Boolean

→ int

→ id

Statement → { Statement* }
→ if (Exp) Statement else Statement
→ while (Exp) Statement
→ System.out.println (Exp);
→ id = Exp ;
→ id [Exp] = Exp;

Exp → Exp op Exp
→ Exp [Exp]
→ Exp . id (ExpList)
→ INTEGER_LITERAL
→ true
→ false
→ this
→ new int [Exp]
→ new id ( )
→ ! Exp
→ (Exp)

ExpList → Exp ExpRest*
→

ExpRest → , Exp

A.3 SAMPLE PROGRAM

```
Class Factorial {
    Public static void main ( String[] a) {
        System.out.println(new Fac().ComputerFac(10));
        }
}
Class Fac {
    Public int ComputerFac(int num) {
        Int num_aux;
        If (num < 1)
            Num_aux = 1;
        Else
            Num_aux = num * (this.ComputeFac(num-1));
            Return num_aux;
    }
}
```

Part II Appended (Intermediate representation trees)

Package Tree;

Abstract class Exp
CONST(int value) //integer constant
NAME(Label label) //symbolic constant as a label in assembly languages
TEMP(Temp.Temp temp) //Temporary, as a register in a real machine
BINOP(int binop, Exp left, Exp right) // The application of binary operator o to operands left and right, where left is evaluated before right. The integer arithmetic operators are PLUS, MINUS, MUL and DIV; the integer bitwise logical operators are AND, OR, and XOR; the integer shift operations are LSHIFT, RSHIFT; the integer arithmetic right-shift is ARSHIFT. The MiniJava language has only one logical operator, but the intermediate language is meant to be independent of any source language; also, the logical operators might be used in implementing other features of MiniJava.
MEM(Exp exp) //The contents of wordSize bytes of memory starting at address exp. When MEM is used as the left child of a MOVE, it means “store,” and “fetch,” others.
CALL(Exp func, ExpList args) //A procedure call: the calling function func with argument list args. The subexpression func is evaluated before the arguments.
ESEQ(Stm stm, Exp exp) //The statement stm is evaluated for side effects, then exp for a result.

Abstract class Stm
MOVE(Exp dst, Exp src) //Evaluate src and move it into temporary dst
EXP(Exp exp) //Evaluate exp and discard the result.
JUMP(Exp exp, Temp.LabelList targets) //Transfer contrl (jump) to address exp. The destination exp may be a literal label, as in NAME(lab), or an address calculated by any other kind of expression, and the targets indicate a list of possible locations.
CJUMP(int relop, Exp left, Exp right, Label iftrue, Label iffalse) //Evaluate left and right in that order, yielding values a and b to be compared by relop. It then jumps to label iftrue if the result is true; iffalse, otherwise.
SEQ(Stm left, Stm right) //The statement left followed by right.
LABEL(Label label) //Define the constant value of label to be the current machine code address.

Other classes:
ExpList(Exp head, ExpList tail)
StmList(Stm head, Stmlist tail)

Other constants:

final static int BINOP.PLUS, BINOP.MINUS, BINOP.MUL, BINOP.DIV, BINOP.AND, BINOP.OR, BINOP.LSHIFT, BINOP.RSHIFT, BINOP.ARSHIFT, BINOP.XOR;

final static int CJUMP.EQ, CJUMP.NE, CJUMP.LT, CJUMP.GT, CJUMP.LE, CJUMP.GE, CJUMP.ULT, CJUMP.ULE, CJUMP.UGT, CJUMP.UGE;

Part III appended: A sample translation package based on above representation trees:

```
Package Translate;
Public abstract class Exp{
    Abstract Tree.Exp unEX(); //Ex stands for an “exression,” represented as a Tree.Exp.
    Abstract Tree.Stm unNX(); //Nx stands for “no result,” represented as a Tree statement.
   Abstract Tree.Stm unCx(Temp.Label t, Temp.Label f); //Cx stands for “conditional,” represented as a function from label-pair to statement. If you pass it a true destination and a false destination, it will make a statement that evaluates some conditionals and then jumps to one of the destinations (the statement will never “fall through”)
}
```